Models of quantum computation and quantum programming languages

Jarosław Adam MISZCZAK
Institute of Theoretical and Applied Informatics,
Polish Academy of Sciences, Bałtycka 5, 44-100 Gliwice, Poland

The goal of the presented paper is to provide an introduction to the basic computational models used in quantum information theory. We review various models of quantum Turing machine, quantum circuits and quantum random access machine (QRAM) along with their classical counterparts. We also provide an introduction to quantum programming languages, which are developed using the QRAM model. We review the syntax of several existing quantum programming languages and discuss their features and limitations.

I. INTRODUCTION

Computational process must be studied using the fixed model of computational device. This paper introduces the basic models of computation used in quantum information theory. We show how these models are defined by extending classical models.

We start by introducing some basic facts about classical and quantum Turing machines. These models help to understand how useful quantum computing can be. It can be also used to discuss the difference between quantum and classical computation. For the sake of completeness we also give a brief introduction to the main results of quantum complexity theory. Next we introduce Boolean circuits and describe the most widely used model of quantum computation, namely quantum circuits. We focus on this model since many presented facts about quantum circuits are used in the following sections. Finally we introduce another model which is more suited for defining programming languages operating on quantum memory — quantum random access machine (QRAM).

We also describe selected examples of the existing quantum programming languages. We start by formulating the requirements which must be fulfilled by any universal quantum programming language. Next we describe languages based on imperative paradigm – QCL (Quantum Computation Language) and LanQ. We also describe recent research efforts focused on implementing languages based on functional paradigm and discuss the advantages of a language based on this paradigm. As the example of functional quantum programming language we present cQPL.

We introduce the syntax and discuss the features of the presented languages. We also point out their weaknesses. For the sake of completeness a few examples of quantum algorithms and protocols are presented. We use these examples to introduce the main features of the presented languages.

Note that we will not discuss problems related to the physical realisation of the described models. We also do not cover the area of quantum error correcting codes, which aims to provide methods for dealing with decoherence in quantum systems. For an introduction to these problems and recent progress in this area see *e.g.* [1, 2].

One should be also aware that the presented overview of existing models of quantum computation is biased towards the models interesting for the development of quantum programming languages. Thus we neglect some models which are not directly related to this area (*e.g.* quantum automata or topological quantum computation).

A. Quantum information theory

Quantum information theory is a new, fascinating field of research which aims to use quantum mechanical description of the system to perform computational tasks. It is based on quantum physics and classical computer science, and its goal is to use the laws of quantum mechanics to develop more powerful algorithms and protocols.

According to the Moore's Law [3, 4] the number of transistors on a given chip is doubled every two years (see Figure 1). Since classical computation has its natural limitations in the terms of the size of computing devices, it is natural to investigate the behaviour of objects in micro scale.

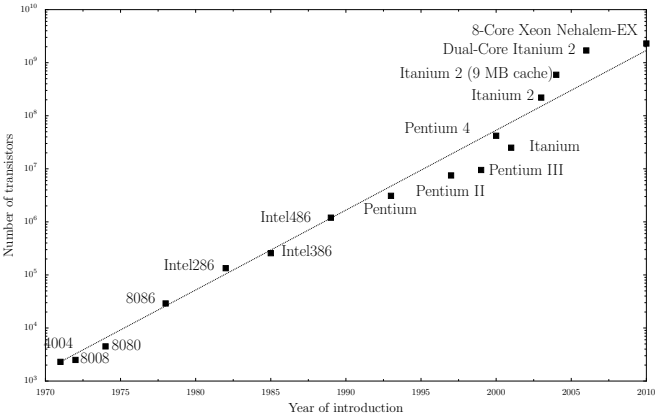


Figure 1. Illustration of Moore's hypothesis. The number of transistors which can be put on a single chip grows exponentially. The squares represent microprocessors introduced by Intel Corporation [4]. The dotted line illustrates the rate of growth, with the number of transistors doubling every two years.

Quantum effects cannot be neglected in microscale and thus they must be taken into account when designing fu-

ture computers. Quantum computation aims not only at taking them into account, but also at developing methods for controlling them. Quantum algorithms and protocols are recipes how one should control quantum system to achieve higher efficiency.

Information processing on quantum computer was first mentioned in 1982 by Feynman [5]. This seminal work was motivated by the fact that simulation of a quantum system on the classical machine requires exponential resources. Thus, if we could control a physical system at the quantum level we should be able to simulate other quantum systems using such machines.

The first quantum protocol was proposed two years later by Bennett and Brassard [6]. It gave the first example of the new effects which can be obtained by using the rules of quantum theory for information processing. In 1991 Ekert described the protocol [7] showing the usage of quantum entanglement [8] in communication theory.

Today we know that thanks to the quantum nature of photons it is possible to create unconditionally secure communication links [9] or send information with the efficiency unachievable using classical carriers. During the last few years quantum cryptographic protocols have been implemented in real-world systems. Quantum key distribution is the most promising application of quantum information theory, if one takes practical applications [10, 11] into account.

On the other hand we know that the quantum mechanical laws of nature allow us to improve the solution of some problems [12–14], construct games [15, 16] and random walks [17, 18] with new properties.

Nevertheless, the most spectacular achievements in quantum information theory up to the present moment are: the quantum algorithm for factoring numbers and calculating discrete logarithms over finite field proposed in the late nineties by Shor [13]. The quantum algorithm solves the factorisation problem in polynomial time, while the best known probabilistic classical algorithm runs in time exponential with respect to the size of input number. Shor's factorisation algorithm is one of the strongest arguments for the conjecture that quantum computers can be used to solve in polynomial time problems which cannot be solved classically in reasonable (*i.e.* polynomial) time.

Taking into account research efforts focused on discovering new quantum algorithms it is surprising that for the last ten years no similar results have been obtained [19, 20]. One should note that there is no proof that quantum computers can actually solve **NP**-complete problems in polynomial time [21, 22]. This proof could be given by quantum algorithms solving in polynomial time problems known to be **NP**-complete such as k-colorability. The complexity of quantum computation remains poorly understood. We do not have much evidence how useful quantum computers can be. Still much remains to be discovered in the area of the relations between quantum complexity classes such as **BQP** and classical complexity classes like **NP**.

B. Progress in quantum algorithms

Due to the slow progress in discovering new quantum algorithms novel methods for studying the impact of quantum mechanics on algorithmic problems were proposed.

The first of these methods aims at applying the rules of quantum mechanics to game theory [16, 23]. Classical games are used to model the situation of conflict between competing agents. The simplest application of quantum games is presented in the form of quantum prisoners dilemma [15]. In this case one can analyse the impact of quantum information processing on classical scenarios. On the other hand quantum games can be also used to analyse typical quantum situations like state estimation and cloning [24].

Quantum walks provide the second promising method for developing new quantum algorithms. Quantum walks are the counterparts of classical random walks [17, 18]. For example, in [25] the quantum algorithm for element distinctness using this method was proposed. It requires $O(n^{2/3})$ queries to determine if the input $\{x_1, \ldots, x_n\}$ consisting of n elements contains two equal numbers. Classical algorithm solving this problem requires $O(n \log n)$ queries. The generalisation of this algorithm, with applications to the problem of subset finding, was described in [26]. Other application of quantum walks include searching algorithms [27] and subset finding problem. It was also shown that quantum walks can be used to perform a universal quantum computation [28, 29]. In [30] the survey of quantum algorithms based on quantum walks is presented. More information concerning recent developments in quantum walks and their applications can be found in [31].

One should note that the development of quantum algorithms is still a very lively area of research [20, 32]. General introduction to quantum algorithms can be found in [33]. The in-depth review of the recent results in the area of quantum algorithms for algebraic problems can be found in [34].

II. COMPUTABILITY

Classically computation can be described using various models. The choice of the model used depends on the particular purpose or problem. Among the most important models of computation we can point:

- **Turing Machine** introduced in 1936 by Turing and used as the main model in complexity theory [35].
- **Random Access Machine** [36, 37] which is the example of register machines; this model captures the main features of modern computers and

provides a theoretical model for programming languages.

- **Boolean circuits** [38] defined in terms of logical gates and used to compute Boolean functions $f : \{0,1\}^m \mapsto \{0,1\}^n$; they are used in complexity theory to study circuit complexity.
- **Lambda calculus** defined by Church [39] and used as the basis for many functional programming languages [40].
- **Universal programming languages** which are probably the most widely used model of computation [41].

It can be shown that all these models are equivalent [35, 38]. In other words the function which is computable using one of these models can be computed using any other model. It is quite surprising since Turing machine is a very simple model, especially when compared with RAM or programming languages.

In particular the model of a multitape Turing machine is regarded as a canonical one. This fact is captured by the Church-Turing hypothesis.

Hypothesis 1 (Church-Turing) *Every function which would be naturally regarded as computable can be computed by a universal Turing machine.*

Although stated as a hypothesis, this thesis is one of the fundamental axioms of modern computer science. A Universal Turing machine is a machine which is able to simulate any other machine. The simplest method for constructing such device is to use the model of a Turing machine with two tapes [35].

Research in quantum information processing is motivated by the extended version of Church-Turing thesis formulated by Deutsch [42].

Hypothesis 2 (Church-Turing-Deutsch) *Every physical process can be simulated by a universal computing device.*

In other words this thesis states that if the laws of physics are used to construct a Turing machine, this model might provide greater computational power when compared with the classical model. Since the basic laws of physics are formulated as quantum mechanics, this improved version of a Turing machine should be governed by the laws of quantum physics.

In this section we review some of these computational models focusing on their quantum counterparts. The discussion of quantum programming languages, which are based on the quantum random access machines (QRAM), is presented in Section III.

We start be recalling the basic facts concerning a Turing machine. This model allows to establish clear notion of computational resources like time and space used during computation. It is also used to define other models introduced in this section precisely.

On the other hand for practical purposes the notion of Turing machine is clumsy. Even for simple algorithms it requires quite complex description of transition rules. Also, programming languages defined using a Turing machine [43], have rather limited set of instructions. Thus we use more sophisticated methods like Boolean circuits and programming languages based on QRAM model.

A. Turing machine

The model of a Turing machine is widely used in classical and quantum complexity theory. Despite its simplicity it captures the notion of computability.

In what follows by *alphabet* $A = \{a_1, \ldots, a_n\}$ we mean any finite set of characters or digits. Elements of A are called letters. Set A^k contains all strings of length k composed from elements of A. Elements of A^k are called *words* and the length of the word w is denoted by $|w|$. The set of all words over A is denoted by A^*. Symbol ϵ is used to denote an empty word. The complement of language $L \subset A^*$ is denoted by $\bar{L}$ and it is the language defined as $\bar{L} = A^* - L$.

1. Classical Turing machine

A Turing machine can operate only using one data structure – the string of symbols. Despite its simplicity, this model can simulate any algorithm with inconsequential loss of efficiency [35]. A Classical Turing machine consists of

- an infinitely long tape containing symbols from the finite alphabet A,
- a head, which is able to read symbols from the tape and write them on the tape,
- memory for storing programme for the machine.

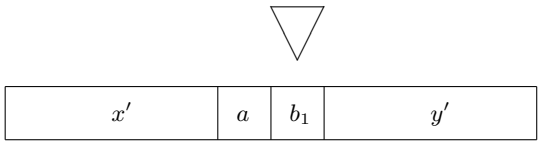
The programme for a Turing machine is given in terms of transition function δ. The schematic illustration of a Turing machine is presented in Figure 2.

Formally, the classical deterministic Turing machine is defined as follows.

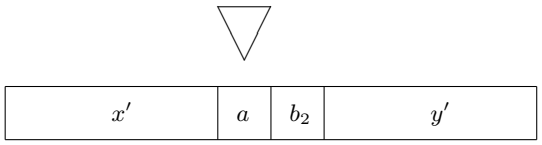
Definition 1 (Deterministic Turing machine) *A deterministic Turing machine M over an alphabet A is a sixtuple $(Q, A, \delta, q_0, q_a, q_r)$, where*

- *Q is the set of internal control states,*
- *$q_0, q_a, q_r \in Q$ are initial, accepting and rejecting states,*
- *$\delta : Q \times A \mapsto Q \times A \times \{-1, 0, 1\}$ is a transition function* i.e. *the programme of a machine.*

By a configuration of machine M we understand a triple (q_i, x, y), $q_i \in Q$, $x, y \in A^*$. This describes a situation where the machine is in the state q_i, the tape contains the word xy and the machine starts to scan the word y. If $x = x'$ and $y = b_1 y'$ we can illustrate this situation as in Figure 2.



(a)Configuration $(q_i, x'a, b_1 y')$



(b)Configuration $(q_j, x', ab_2 y')$

Figure 2. Computational step of the Turing machine. Configuration $(q_i, x'a, b_1 y')$ is presented in (a). If the transition function is defined such that $\delta(q_i, b_1) = (q_2, b_2, -1)$ this computational step leads to configuration $(q_j, x', ab_2 y')$ (see (b)).

The transition from the configuration c_1 to the configuration c_2 is called a computational step. We write $c \vdash c'$ if δ defines the transition from c to c'. In this case c' is called the successor of c.

A Turing machine can be used to compute values of functions or to decide about input words. The computation of a machine with input $w \in A^*$ is defined as a sequence of configurations $c_0, c_1, c_2, \ldots$, such that $c_0 = (q_i, \epsilon, w)$ and $c_i \vdash c_{i+1}$. We say that computation halts if some c_i has no successor or for configuration c_i, the state of the machine is q_a (machine accepts input) or q_r (machine rejects input).

The computational power of the Turing machine has its limits. Let us define two important classes of languages.

Definition 2 *A set of words $L \in A^*$ is a recursively enumerable language if there exists a Turing machine accepting input w iff $w \in L$.*

Definition 3 *A set of words $L \in A^*$ is a recursive language if there exists a Turing machine M such that*

- *M accepts w iff $w \in L$,*
- *M halts for any input.*

The computational power of the Turing machine is limited by the following theorem.

Theorem 1 *There exists a language H which is recursively enumerable but not recursive.*

Language H used in the above theorem is defined in halting problem [35]. It consists of all words composed of words encoding Turing machines and input words for these machines, such that a particular machine halts on a given word. A universal Turing machine can simulate any machine, thus for a given input word encoding machine and input for this machine we can easily perform the required computation.

A deterministic Turing machine is used to measure time complexity of algorithms. Note that if for some language there exists a Turing machine accepting it, we can use this machine as an algorithm for solving this problem. Thus we can measure the running time of the algorithm by counting the number of computational steps required for Turing machine to output the result.

The time complexity of algorithms can be described using the following definition.

Definition 4 *Complexity class* $\mathbf{TIME}(f(n))$ *consists of all languages L such that there exists a deterministic Turing machine running in time $f(n)$ accepting input w iff $w \in L$.*

In particular complexity class $\mathbf{P}$ defined as

$$\mathbf{P} = \bigcup_k \mathbf{TIME}(n^k), \tag{1}$$

captures the intuitive class of problems which can be solved *easily* on a Turing machine.

2. Nondeterministic and probabilistic computation

Since one of the main features of quantum computers is their ability to operate on the superposition of states we can easily extend the classical model of a probabilistic Turing machine and use it to describe quantum computation. Since in general many results in the area of algorithms complexity are stated in the terms of a nondeterministic Turing machine we start by introducing this model.

Definition 5 (Nondeterministic Turing machine) *A nondeterministic Turing machine M over an alphabet A is a sixtuple $(Q, A, \delta, q_0, q_a, q_r)$, where*

- *Q is the set of internal control states,*
- *$q_0, q_a, q_r \in Q$ are initial, accepting and rejecting states,*
- *$\delta \subset Q \times A \times Q \times A \times \{-1, 0, 1\}$ is a relation.*

The last condition in the definition of a nondeterministic machine is the reason for its power. It also requires to change the definition of acceptance by the machine.

We say that a nondeterministic Turing machine accepts input w if, for some initial configuration (q_i, ϵ, w), computation leads to configuration (q_a, a_1, a_2) for some

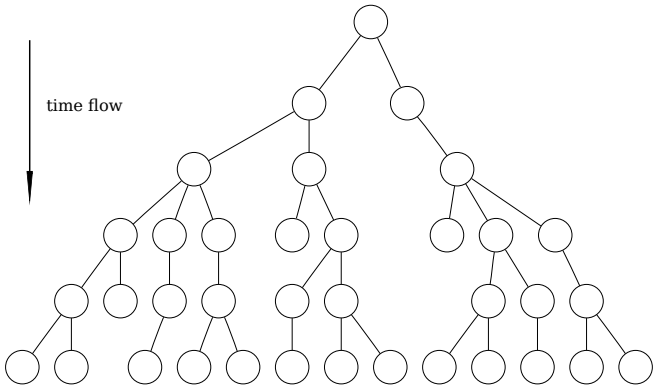


Figure 3. Schematic illustration of the computational paths of a nondeterministic Turing machine [35]. Each circle represents the configuration of the machine. The machine can be in many configurations simultaneously.

words a_1 and a_2. Thus a nondeterministic machine accepts the input if there exists some computational path defined by transition relation δ leading to an accepting state q_a.

The model of a nondeterministic Turing machine is used to define complexity classes **NTIME**.

Definition 6 *Complexity class* $\mathbf{NTIME}(f(n))$ *consists of all languages* L *such that there exists a nondeterministic Turing machine running in time* $f(n)$ *accepting input* w *iff* $w \in L$.

The most prominent example of these complexity classes is **NP**, which is the union of all $\mathbf{NTIME}(n^k)$, *i.e.*

$$\mathbf{NP} = \bigcup_k \mathbf{NTIME}(n^k). \tag{2}$$

A nondeterministic Turing machine is used as a theoretical model in complexity theory. However, it is hard to imagine how such device operates. One can illustrate the computational path of a nondeterministic machine as in Figure 3 [35].

Since our aim is to provide the model of a physical device we restrict ourselves to more realistic model. We can do that by assigning to each element of relation a number representing probability. In this case we obtain the model of a probabilistic Turing machine.

Definition 7 (Probabilistic Turing machine) *A probabilistic Turing machine* M *over an alphabet* A *is a sixtuple* $(Q, A, \delta, q_0, q_a, q_r)$*, where*

- Q *is the set of internal control states,*
- $q_0, q_a, q_r \in Q$ *are initial, accepting and rejecting states,*
- $\delta : Q \times A \times Q \times A \times \{-1, 0, 1\} \mapsto [0, 1]$ *is a transition probability function* i.e.

$$\sum_{(q_2,a_2,d)\in Q\times A\times\{-1,0,1\}} \delta(q_1, a_1, q_2, a_2, d) = 1. \tag{3}$$

For a moment we can assume that the probabilities of transition used by a probabilistic Turing machine can be represented only by rational numbers. We do this to avoid problems with machines operating on arbitrary real numbers. We will address this problem when extending the above definition to the quantum case.

The time complexity of computation can be measured in terms of the number of computational steps of the Turing machine required to execute a programme. Among important complexity classes we have chosen to point out:

- **P** – the class of languages for which there exists a deterministic Turing machine running in polynomial time,
- **NP** – the class of languages for which there exists a nondeterministic Turing machine running in polynomial time,
- **RP** – the class of languages L for which there exists a probabilistic Turing machine M such that: M accepts input w with probability at least $\frac{1}{2}$ if $w \in L$ and always rejects w if $w \notin L$,
- **coRP** – the class of languages L for which $\bar{L}$ is in **RP**,
- **ZPP** – $\mathbf{RP} \cap \mathbf{coRP}$.

More examples of interesting complexity classes and computational problems related to them can be found in [44].

3. Quantum Turing machine

A quantum Turing machine was introduced by Deutsch [42]. This model is equivalent to a quantum circuit model [45, 46]. However, it is very inconvenient for describing quantum algorithms since the state of a head and the state of a tape are described by state vectors.

A quantum Turing machine consists of

- Processor: M 2-state observables $\{n_i | i \in_{\mathbb{Z}_M}\}$.
- Memory: infinite sequence of 2-state observables $\{m_i | i \in \mathbb{Z}\}$.
- Observable x, which represents the address of the current head position.

The state of the machine is described by the vector $|\psi(t)\rangle = |x; n_0, n_1, \ldots; m\rangle$ in the Hilbert space $\mathcal{H}$ associated with the machine.

At the moment $t = 0$ the state of the machine is described by the vectors $|\psi(0)\rangle = \sum_m a_m |0; 0, \ldots, 0; \ldots, 0, 0, 0, \ldots\rangle$ such that

$$\sum_i |a_i|^2 = 1. \tag{4}$$

The evolution of the quantum Turing machine is described by the unitary operator U acting on $\mathcal{H}$.

A classical probabilistic (or nondeterministic) Turing machine can be described as a quantum Turing machine such that, at each step of its evolution, the state of the machine is represented by the base vector.

The formal definition of the quantum Turing machine was introduced in [21].

It is common to use real numbers as amplitudes when describing the state of quantum systems during quantum computation. To avoid problems with an arbitrary real number we introduce the class of numbers which can be used as amplitudes for amplitude transition functions of the quantum Turing machine.

Let us denote by $\widetilde{\mathbb{C}}$ the set of complex numbers $c \in \mathbb{C}$, such that there exists a deterministic Turing machine, which allows to calculate $\mathrm{Re}\,(c)$ and $\mathrm{Im}\,(c)$ with accuracy $\frac{1}{2^n}$ in time polynomial in n.

Definition 8 (Quantum Turing Machine) *A quantum Turing machine (QTM) M over an alphabet A is a sixtuple $(Q, A, \delta, q_0, q_a, q_r)$, where*

- *Q is the set of internal control states,*
- *$q_0, q_a, q_r \in Q$ are initial, accepting and rejecting states,*
- *$\delta : Q \times A \times Q \times A \times \{-1, 0, 1\} \mapsto \widetilde{\mathbb{C}}$ is a transition amplitude function* i.e.

$$\sum_{(q_2,a_2,d)\in Q\times A\times\{-1,0,1\}} |\delta(q_1, a_1, q_2, a_2, d)|^2 = 1. \qquad (5)$$

Reversible classical Turing machines (*i.e.*Turing machines with reversible transition function) can be viewed as particular examples of quantum machines. Since any classical algorithm can be transformed into reversible form, it is possible to simulate a classical Turing machine using quantum Turing machine.

4. Quantum complexity

Quantum Turing machine allows for rigorous analysis of algorithms. This is important since the main goal of quantum information theory is to provide some gain in terms of speed or memory with respect to classical algorithms. It should be stressed that at the moment no formal proof has been given that a quantum Turing machine is more powerful than a classical Turing machine [22].

In this section we give some results concerning quantum complexity theory. See also [21, 47] for a introduction to this subject.

In analogy to classical case it is possible to define complexity classes for the quantum Turing machine. The most important complexity class is this case is **BQP**.

Definition 9 *Complexity class* **BQP** *contains languages L for which there exists a quantum Turing machine running in polynomial time such that, for any input word x, this word is accepted with probability at least $\frac{3}{4}$ if $x \in L$ and is rejected with probability at least $\frac{3}{4}$ if $x \notin L$.*

Class **BQP** is a quantum counterpart of the classical class **BPP**.

Definition 10 *Complexity class* **BPP** *contains languages L for which there exists a nondeterministic Turing machine running in polynomial time such that, for any input word x, this word is accepted with probability at least $\frac{3}{4}$ if $x \in L$ and is rejected with probability at least $\frac{3}{4}$ if $x \notin L$.*

Since many results in complexity theory are stated in terms of oracles, we define an oracle as follows.

Definition 11 *An oracle or black box is an imaginary machine which can decide certain problems in a single operation.*

We use notation $\mathbf{A^B}$ to describe the class of problems solvable by an algorithm in class **A** with an oracle for the language **B**.

It was shown [21] that the quantum complexity classes are related as follows.

Theorem 2 *Complexity classes fulfil the following inequality*

$$\mathbf{BPP} \subseteq \mathbf{BQP} \subseteq \mathbf{P^{\#P}}. \qquad (6)$$

Complexity class $\#\mathbf{P}$ consists of problems of the form *compute f(x), where f is the number of accepting paths of an* **NP** *machine*. For example problem $\#\mathbf{SAT}$ formulated below is in $\#\mathbf{P}$.

Problem 1 ($\#$SAT) *For a given Boolean formula, compute how many satisfying true assignments it has.*

Complexity class $\mathbf{P^{\#P}}$ consists of all problems solvable by a machine running in polynomial time which can use oracle for solving problems in $\#\mathbf{P}$.

Complexity ZOO [44] contains the description of complexity classes and many famous problems from complexity theory. The complete introduction to the complexity theory can be found in [35]. Theory of **NP**-completeness with many examples of problems from this class is presented in [48].

Many important results and basic definitions concerning quantum complexity theory can be found in [21]. The proof of equivalence between quantum circuit and quantum Turing machine was given in [45]. An interesting discussion of quantum complexity classes and relation of **BQP** class to classical classes can be found in [22].

B. Quantum computational networks

After presenting the basic facts about Turing machines we are ready to introduce more usable models of computing devices. We start by defining Boolean circuits and extending this model to the quantum case.

1. Boolean circuits

Boolean circuits are used to compute functions of the form

$$f : \{0,1\}^m \mapsto \{0,1\}^n. \tag{7}$$

Basic gates (functions) which can be used to define such circuits are

- $\wedge : \{0,1\}^2 \mapsto \{0,1\}$, $\wedge(x,y) = 1 \Leftrightarrow x = y = 1$ (*logical and*),
- $\vee : \{0,1\}^2 \mapsto \{0,1\}$, $\vee(x,y) = 0 \Leftrightarrow x = y = 0$ (*logical or*),
- $\sim$: $\{0,1\} \mapsto \{0,1\}$, $\sim(x) = 1 - x$ (*logical not*).

The set of gates is called universal if all functions $\{0,1\}^n \mapsto \{0,1\}$ can be constructed using the gates from this set. It is easy to show that the set of functions composed of $\sim$, $\vee$ and $\wedge$ is universal. Thus it is possible to compute any functions $\{0,1\}^n \mapsto \{0,1\}^m$ using only these functions. The full characteristic of universal sets of functions was given by Post in 1949 [49].

Using the above set of functions a Boolean circuit is defined as follows.

Definition 12 (Boolean circuit) *A Boolean circuit is an acyclic direct graph with nodes labelled by input variables, output variables or logical gates $\vee$, $\wedge$ or $\sim$.*

Input variable node has no incoming arrow while output variable node has no outcoming arrows. The example of a Boolean circuit computing the sum of bits x_1 and x_2 is given in Figure 4.

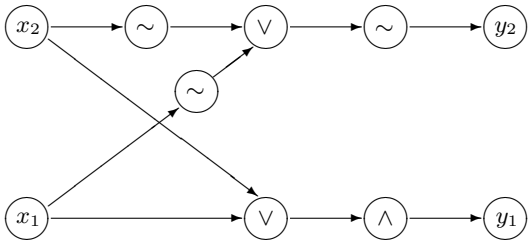


Figure 4. The example of a Boolean circuit computing the sum of bits x_1 and x_2 [50]. Nodes labelled x_1 and x_2 represent input variables and nodes labelled y_1 and y_2 represent output variables.

Note that in general it is possible to define a Boolean circuit using different sets of elementary functions. Since functions $\vee$, $\wedge$ and $\sim$ provide a universal set of gates we defined Boolean circuit using these particular functions.

Function $f : \{0,1\}^m \mapsto \{0,1\}$ is defined on the binary string of arbitrary length. Let $f_n : \{0,1\}^m \mapsto \{0,1\}^n$ be a restriction of f to $\{0,1\}^n$. For each such restriction there is a Boolean circuit C_n computing f_n. We say that $C_0, C_1, C_2, \ldots$ is a *family of Boolean circuits* computing f.

Note that any binary language $L \subset \{0,1\}^*$ can be accepted by some family of circuits. But since we need to know the value of f_n to construct a circuit C_n such family is not an algorithmic device at all. We can state that there exists a family accepting the language, but we do not know how to build it [35].

To show how Boolean circuits are related to Turing machines we introduce uniformly generated circuits.

Definition 13 *We say that language $L \in A^*$ has uniformly polynomial circuits if there exists a Turing machine M that an input $\underbrace{1\ldots1}_{n}$ outputs the graph of circuit C_n using space $O(\log n)$, and the family $C_0, C_1, \ldots$ accepts L.*

The following theorem provides a link between uniformly generated circuits and Turing machines.

Theorem 3 *A language L has uniformly polynomial circuit iff $L \in \mathbf{P}$.*

Quantum circuits model is an analogous to uniformly polynomial circuits. They can be introduced as the straightforward generalisation of reversible circuits.

2. Reversible circuits

The evolution of isolated quantum systems is described by a unitary operator U. The main difference with respect to classical evolution is that this type of evolution is *reversible*.

Before introducing a quantum circuit we define a reversible Boolean circuit

Definition 14 (Reversible gate) *A classical reversible function (gate) $\{0,1\}^m \mapsto \{0,1\}^m$ is a permutation.*

Definition 15 *A reversible Boolean circuit is a Boolean circuit composed of reversible gates.*

The important fact expressed by the following theorem allows us to simulate any classical computation on a quantum machine described using a reversible circuit

Theorem 4 *All Boolean circuits can be simulated using reversible Boolean circuits.*

Like in the case of nonreversible circuit one can introduce the universal set of functions for reversible circuits.

The important example of a gate universal for reversible Boolean circuits is a Toffoli gate. The graphical representation of this gate is presented in Figure 5. The following theorem was proved by Toffoli [52].

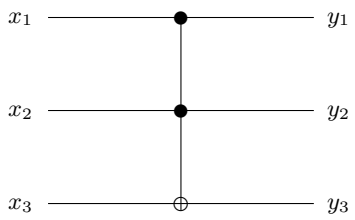


Figure 5. Classical Toffoli gate is universal for reversible circuits. It was also used to provide the universal set of quantum gates [51].

Theorem 5 *A Toffoli gate is a universal reversible gate.*

As we will see in the following section it is possible to introduce two-bit quantum gates which are universal for quantum circuits. This is impossible in the classical case and one needs at least a three-bit gate to construct the universal set of reversible gates.

In particular, any reversible circuit is automatically a quantum circuit. However, quantum circuits offer much more diversity in terms of the number of allowed operations.

3. Quantum circuits

The computational process of the quantum Turing machine is complicated since data as well as control variables can be in a superposition of base states. To provide more convenient method of describing quantum algorithms one can use a quantum circuits model. This model is sometimes called a *quantum gate array* model.

Quantum circuits model was first introduced by Deutsch in [51] and it is the most commonly used notation for quantum algorithms. It is much easier to imagine than the quantum Turing machine since the control variables (executed steps and their number) are classical. There are only quantum data (*e.g.* qubits or qudits and unitary gates) in a quantum circuit.

A quantum circuit consists of the following elements (see Table II):

- the finite sequence of *wires* representing qubits or sequences of qubits (quantum registers),
- quantum gates representing elementary operations from the particular set of operations implemented on a quantum machine,
- measurement gates representing a measurement operation, which is usually executed as the final step of a quantum algorithm. It is commonly assumed that it is possible to perform the measurement on each qubit in canonical basis $\{|0\rangle, |1\rangle\}$ which corresponds to the measurement of the S_z observable.

The concept of a quantum circuit is the natural generalisation of acyclic logic circuits studied in classical computer science. Quantum gates have the same number of inputs as outputs. Each n-qubit quantum gate represents the 2^n-dimensional unitary operation of the group $SU(2^n)$, *i.e.* generalised rotation in a complex Hilbert space.

The main advantage of this model is its simplicity. It also provides very convenient representation of physical evolution in quantum systems.

From the mathematical point of view quantum gates are unitary matrices acting on n-dimensional Hilbert space. They represent the evolution of an isolated quantum system [53].

The problem of constructing new quantum algorithms requires more careful study of operations used in quantum circuit model. In particular we are interested in efficient decomposition of quantum gates into elementary operations.

We start by providing basic charactersistic of unitary matrices [53, 54]

Theorem 6 *Every unitary 2×2 matrix $G \in U(2)$ can be decomposed using elementary rotations as*

$$G = \Phi(\delta) R_z(\alpha) R_y(\theta) R_z(\beta) \tag{8}$$

where

$$\Phi(\xi) = \begin{pmatrix} e^{i\xi} & 0 \\ 0 & e^{i\xi} \end{pmatrix},$$

$$R_y(\xi) = \begin{pmatrix} \cos(\xi/2) & \sin(\xi/2) \\ -\sin(\xi/2) & \cos(\xi/2) \end{pmatrix},$$

and

$$R_z(\xi) = \begin{pmatrix} e^{i\frac{\xi}{2}} & 0 \\ 0 & e^{-i\frac{\xi}{2}} \end{pmatrix}.$$

We introduce the definition of quantum gates as stated in [50].

Definition 16 *A quantum gate U acting on m qubits is a unitary mapping on $\mathbb{C}^{2^m} \equiv \underbrace{\mathbb{C}^2 \otimes \ldots \otimes \mathbb{C}^2}_{m \text{ times}}$*

$$U : \mathbb{C}^{2^m} \mapsto \mathbb{C}^{2^m}, \tag{9}$$

which operates on the fixed number of qubits.

Formally, a quantum circuit is defined as the unitary mapping which can be decomposed into the sequence of elementary gates.

Definition 17 *A quantum circuit on m qubits is a unitary mapping on $\mathbb{C}^{2^m}$, which can be represented as a concatenation of a finite set of quantum gates.*

Any reversible classical gate is also a quantum gate. In particular logical gate $\sim$ (negation) is represented by quantum gate NOT, which is realized by σ_x Pauli matrix.

As we know any Boolean circuit can be simulated by a reversible circuit and thus any function computed by

a Boolean circuit can be computed using a quantum circuit. Since a quantum circuit operates on a vector in complex Hilbert space it allows for new operations typical for this model.

The first example of quantum gate which has no classical counterpart is $\sqrt{NOT}$ gate. It has the following property

$$\sqrt{NOT}\sqrt{NOT} = NOT, \tag{10}$$

which cannot be fulfilled by any classical Boolean function $\{0,1\} \mapsto \{0,1\}$. Gate $\sqrt{N}$ is represented by the unitary matrix

$$\sqrt{NOT} = \frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}. \tag{11}$$

Another example is Hadamard gate H. This gate is used to introduce the superposition of base states. It acts on the base state as

$$H|0\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right), H|1\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle - |1\rangle\right). \tag{12}$$

If the gate G is a quantum gate acting on one qubit it is possible to construct the family of operators acting on many qubits. The particularly important class of multiqubit operations is the class of controlled operations.

Definition 18 (Controlled gate) *Let G be a 2×2 unitary matrix representing a quantum gate. Operator*

$$|1\rangle\langle 1| \otimes G + |0\rangle\langle 0| \otimes \mathbb{I} \tag{13}$$

acting on two qubits, is called a controlled-G gate.

Here $A \otimes B$ denotes the tensor product of gates (unitary operator) A and B, and $\mathbb{I}$ is an identity matrix. If in the above definition we take $G = NOT$ we get

$$|1\rangle\langle 1| \otimes \sigma_x + |0\rangle\langle 0| \otimes \mathbb{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{14}$$

which is the definition of $CNOT$(controlled-NOT) gate. This gate can be used to construct the universal set of quantum gates. This gate also allows to introduce entangled states during computation

$$\begin{aligned} CNOT(H \otimes \mathbb{I})|00\rangle &= CNOT\frac{1}{\sqrt{2}}\left(|00\rangle + |10\rangle\right) \\ &= \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right) \end{aligned}$$

The classical counterpart of $CNOT$ gate is XOR gate.

Other examples of single-qubit and two-qubit quantum gates are presented in Table II. In Figure 6 a quantum circuit for quantum Fourier transform on three qubits is presented.

One can extend Definition 18 and introduce quantum gates with many controlled qubits.

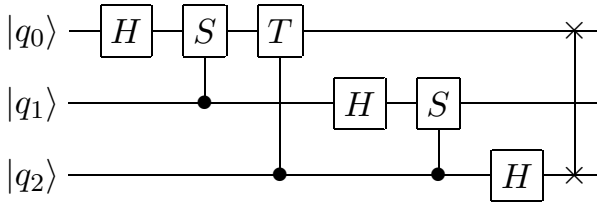


Figure 6. Quantum circuit representing quantum Fourier transform for three qubits. Elementary gates used in this circuit are described in Table II.

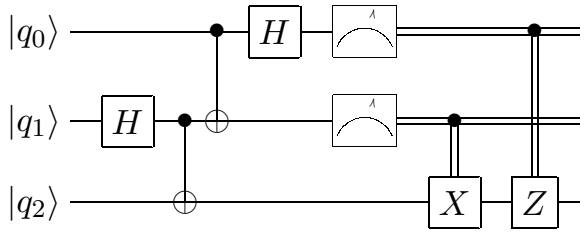


Figure 7. Circuit for quantum teleportation. Double lines represent the operation which is executed depending on the classical data obtained after the measurement on a subsystem.

Definition 19 *Let G be a 2×2 unitary matrix. Quantum gate defined as*

$$|\underbrace{1\ldots1}_{n-1}\rangle\langle\underbrace{1\ldots1}_{n-1}| \otimes G + \sum_{l\neq\underbrace{1\ldots1}_{n-1}} |l\rangle\langle l| \otimes \mathbb{I} \tag{15}$$

is called $(n-1)$-controlled G gate. We denote this gate by $\wedge_{n-1}(G)$.

This gate $\wedge_{n-1}(G)$ is sometimes referred to as a generalised Toffoli gate or a Toffoli gate with m controlled qubits. Graphical representation of this gate is presented in Figure 8.

The important feature of quantum circuits is expressed by the following universality property [54].

Theorem 7 *The set of gates consisting of all one-qubit gates $U(2)$ and one two-qubit CNOT gate is universal in the sense that any n-qubit operation can be expressed as the composition of these gates.*

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Table I. Logical values for XOR gate. Quantum CNOT gate computes value of x_1 XOR x_2 in the first register and stores values of x_2 in the second register.

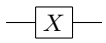
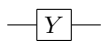
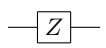
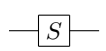
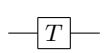
The name of the gate	Graphical representation	Mathematical form
Hadamard		$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
Pauli X		$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
Pauli Y		$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$
Pauli Z		$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
Phase		$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
$\pi/8$		$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$
CNOT		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$
SWAP		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
Measurement		$\left\{\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}\right\}$
qubit		wire $\equiv$ single qubit
n qubits		wire representing n qubits
classical bit		double wire $\equiv$ single bit

Table II. Basic gates used in quantum circuits with their graphical representation and mathematical form. Note that measurement gate is represented in Kraus form, since it is the example of non-unitary quantum evolution.

Note that, in contrast to the classical case, where one needs at least three-bit gates to construct a universal set, quantum circuits can be simulated using one two-qubit universal gate.

In order to implement a quantum algorithm one has to decompose many qubit quantum gates into elementary gates. It has been shown that almost any n-qubit quantum gate ($n \geq 2$) can be used to the build a universal set of gates [55] in the sense that any unitary operation on the arbitrary number of qubits can be expressed as the composition of gates from this set. In fact the set consisting of two-qubit exclusive-or (XOR) quantum gate and all single-qubit gates is also universal [54].

Let us assume that we have the set of gates containing only CNOT and one-qubit gates. In [56] theoretical lower bound for the number of gates required to simulate a circuit using these gates was derived. The efficient method of elementary gates sequence synthesis for an arbitrary unitary gate was presented in [57].

Theorem 8 (Shende-Markov-Bullock) *Almost all n-qubit operators cannot be simulated by a circuit with fewer than $\lceil \frac{1}{4}[4^n - 3n - 1] \rceil$ CNOT gates.*

In [58] the construction providing the efficient way of implementing arbitrary quantum gates was described.

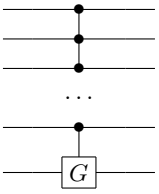


Figure 8. Generalised quantum Toffoli gate acting on n qubits. Gate G is controlled by the state of $n-1$ qubits according to Definition 19.

The resulting circuit has complexity $O(4^n)$ which coincides with lower bound from Theorem 8.

It is useful to provide more details about the special case, when one uses gates with many controlled and one target qubits. The following results were proved in [54].

Theorem 9 *For any single-qubit gate U the gate $\wedge_{n-1}(U)$ can be simulated in terms of $\Theta(n^2)$ basic operations.*

In many situations it is useful to construct a circuit which approximates the required circuit. We say that quantum circuits approximate other circuits with accuracy ε if the distance (in terms of Euclidean norm) between unitary transformations associated with these circuits is at most ε [54].

Theorem 10 *For any single-qubit gate U and $\varepsilon > 0$ gate $\wedge_{n-1}(U)$ can be approximated with accuracy ε using $\Theta(n \log \frac{1}{\varepsilon})$ basic operations.*

Note that the efficient decomposition of a quantum circuit is crucial in physical implementation of quantum information processing. In particular case decomposition can be optimised using the set of elementary gates specific for target architecture. CNOT gates are of big importance since they allow to introduce entangled states during computation. It is also hard to physically realise CNOT gate since one needs to control physical interaction between qubits.

One should also note that for some classes of quantum circuits it is possible to construct their classical counterparts, which can be used to simulate quantum computation performed by these circuits efficiently. The most notable class having this property is a class of circuits CHP class, which consists of stabilizer circuits, *i.e.* circuits consisting solely of CNOT, Hadamard and phase gates [59]. This property is known as so called Gottesman-Knill theorem.

Theorem 11 (Gottesman-Knill) *Any stabilizer circuit can be efficiently simulated on a classical machine.*

It is worth noting that gates used to construct stabilizer circuits do not provide an universal set of gates. Nevertheless, such circuits can produce highly entangled states.

C. Random access machines

Quantum circuit model does not provide a mechanism for controlling with classical machine the operations on quantum memory. Usually quantum algorithms are described using mathematical representation, quantum circuits and classical algorithms [60]. The model of quantum random access machine is built on an assumption that the quantum computer has to be controlled by a classical device [61]. Schematic presentation of such architecture is provided in Figure 9.

Quantum random access machine is interesting for us since it provides a convenient model for developing quantum programming languages. However, these languages are our main area of interest. We see no point in providing the detailed description of this model as it is given in [61] together with the description of hybrid architecture used in quantum programming.

1. Classical RAM model

The classical model of random access machine (RAM) is the example of more general register machines [36, 37].

The random access machine consists of an unbounded sequence of memory registers and a finite number of arithmetic registers. Each register may hold an arbitrary integer number. The programme for the RAM is a finite sequence of instructions $\Pi = (\pi_1, \ldots, \pi_n)$. At each step of execution register i holds an integer r_i and the machine executes instruction π_κ, where κ is the value of the programme counter. Arithmetic operations are allowed to compute the address of a memory register.

Despite the difference in the construction between a Turing machine and RAM, it can be easily shown that a Turing machine can simulate any RAM machine with polynomial slow-down only [35].

It is worth noting that programming languages can be defined without using RAM model. Interesting programming language for a Turing machine $\mathcal{P}''$, providing the minimal set of instructions, was introduced by Böhm in [43].

2. Quantum RAM model

Quantum random access machine (QRAM) model is the extension of the classical RAM. QRAM can exploit quantum resources and, at the same time, can be used to perform any kind of classical computation. It allows us to control operations performed on quantum registers and provides the set of instructions for defining them.

Recently a new model of sequential quantum random machine (SQRAM) has been proposed. Instruction set for this model and compilation of high-level languages is discussed in [62]. However, it is very similar to QRAM model.

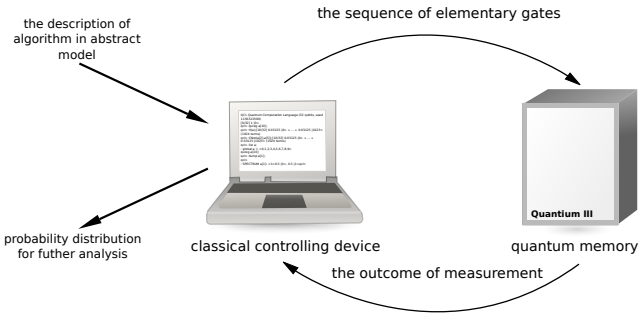


Figure 9. The model of classically controlled quantum machine [61]. Classical computer is responsible for performing unitary operations on quantum memory. The results of quantum computation are received in the form of measurement results.

The quantum part of QRAM model is used to generate probability distribution. This is achieved by performing measurement on quantum registers. The obtained probability distribution has to be analysed using a classical computer.

3. Quantum pseudocode

Quantum algorithms are, in most of the cases, described using the mixture of quantum gates, mathematical formulas and classical algorithms. The first attempt to provide a uniform method of describing quantum algorithms was made in [63], where the author introduced a high-level notation based on the notation known from computer science textbooks [64].

In [60] the first formalised language for description of quantum algorithms was introduced. Moreover, it was tightly connected with the model of quantum machine called quantum random access machine (QRAM).

Quantum pseudocode proposed by Knill [60] is based on conventions for classical pseudocode proposed in [64, Chapter 1]. Classical pseudocode was designed to be readable by professional programmers, as well as people who had done a little programming. Quantum pseudocode introduces operations on quantum registers. It also allows to distinguish between classical and quantum registers.

Quantum registers are distinguished by underlining them. They can be introduced by applying quantum operations to classical registers or by calling a subroutine which returns a quantum state. In order to convert a quantum register into a classical register measurement operation has to be perfomed.

The example of quantum pseudocode is presented in Listing 1. It shows the main advantage of QRAM model over quantum circuits model – the ability to incorporate classical control into the description of quantum algorithm.

Operation $\mathcal{H}(\underline{a_i})$ executes a quantum Hadamard gate

Procedure: FOURIER$(\underline{a}, d)$
Input: A quantum register $\underline{a}$ with d qubits. Qubits are-numbered from 0 to $d-1$.
Output: The amplitudes of $\underline{a}$ are Fourier transformed over $\mathbb{Z}_{2^d}$.

```
C: assign value to classical variable
ω ← e^{i2π/2^d}
C: perform sequence of gates
for i = d − 1 to i = 0
  for j = d − 1 to j = i + 1
  if a_j then R_{ω^{2^{d−i−1+j}}}(a_i)
    C: number of loops executing phase
    C: depends on the required accuracy
    C: of the procedure
  H(a_i)

C: change the order of qubits
for j = 0 to j = d/2 − 1
  SWAP(a_j, a_{d−a−j})
```

Listing 1. Quantum pseudocode for quantum Fourier transform on d qubits. Quantum circuit for this operation with $d = 3$ is presented in Figure 6.

on a quantum register $\underline{a_i}$ and $\mathcal{SWAP}(\underline{a_i}, \underline{a_j})$ performs SWAP gate between $\underline{a_i}$ and $\underline{a_j}$. Operation $\mathcal{R}_\phi(\underline{a_i})$ that executes a quantum gate $R(\phi)$ is defined as

$$R(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}, \tag{16}$$

on the quantum register $\underline{a_i}$. Using conditional contruction

if $\underline{a_j}$ **then** $\mathcal{R}_\phi(\underline{a_i})$

it is easy to define controlled phase shift gate (see Definition 19). Similar construction exists in QCL quantum programming language described in Section III.

The measurement of a quantum register can be indicated using an assignement $a_j \leftarrow \underline{a_j}$.

4. Quantum programming environment

Since the main aim of this paper is to present the advantages and limitations of high-level quantum programming languages, we need to explain how these languages are related to quantum random access machine. Thus as the summary of this section we present the overview of an architecture for quantum programming, which is based on the QRAM model.

The architecture proposed in [65, 66] is designed for transforming a high-level quantum programming language into the technology-specific implementation set of operations. This architecture is composed of four layers:

- **High level programming language** providing high-level mechanisms for performing useful

quantum computation; this language should be independent from particular physical implementation of quantum computing.

- **Compiler of this language** providing architecture independent optimisation; also compilation phase can be used to handle quantum error correction required to perform useful quantum computation.
- **Quantum assembly language (QASM)** – assembly language extended by the set of instructions used in the quantum circuit model.
- **Quantum physical operations language (QCPOL)**, which describes the execution of quantum programme in a hardware-dependent way; it includes physical operations and it operates on the universal set of gates optimal for a given physical implementation.

The authors of [65, 66] do not define a specific high-level quantum programming language. They point out, however, that existing languages, mostly based on Dirac notation, do not provide the sufficient level of abstraction. They also stress, following [67], that it should have the basic set of features. We will discuss these basic requirements in detail in Section III. At the moment quantum assembly language (QASM) is the most interesting part of this architecture, since it is tightly connected to the QRAM model.

QASM should be powerful enough for representing high level quantum programming language and it should allow for describing any quantum circuit. At the same time it must be implementation-independent so that it could be used to optimise the execution of the programme with respect to different architectures.

QASM uses qubits and cbits (classical bit) as basic units of information. Quantum operations consist of unitary operations and measurement. Moreover, each unitary operator is expressed in terms of single qubit gates and CNOT gates.

In the architecture proposed in [66] each single-qubit operation is stored as the triple of rationals. Each rational multiplied by π represents one of three Euler-angles, which are sufficient to specify one-qubit operation.

III. QUANTUM PROGRAMMING LANGUAGES

Quantum algorithms [12, 14, 68, 69] and communication protocols [6, 70, 71] are described using a language of quantum circuits [53]. While this method is convenient in the case of simple algorithms, it is very hard to operate on compound or abstract data types like arrays or integers using this notation [19, 72].

This lack of data types and control structures motivated the development of quantum pseudocode [60, 73] and various quantum programming languages [61, 66, 74–77].

Several languages and formal models were proposed for the description of quantum computation process. The most popular of them is quantum circuit model [51], which is tightly connected to the physical operations implemented in the laboratory. On the other hand the model of quantum Turing machine is used for analysing the complexity of quantum algorithms [21].

Another model used to describe quantum computers is Quantum Random Access Machine (QRAM). In this model we have strictly distinguished the quantum part performing computation and the classical part, which is used to control computation. This model is used as a basis for most quantum programming languages [78–80]. Among high-level programming languages designed for quantum computers we can distinguish imperative and functional languages.

At the moment of writing this paper the most advanced imperative quantum programming language is Quantum Computation Language (QCL) designed and implemented by Ömer [61, 81, 82]. QCL is based on the syntax of C programming language and provides many elements known from classical programming languages. The interpreter is implemented using simulation library for executing quantum programmes on classical computer, but it can be in principle used as a code generator for classical machine controlling a quantum circuit.

Along with QCL several other imperative quantum programming languages were proposed. Notably Q Language developed by Betteli [67, 74] and libquantum [83] have the ability to simulate noisy environment. Thus, they can be used to study decoherence and analyse the impact of imperfections in quantum systems on the accuracy of quantum algorithms.

Q Language [84] is implemented as a class library for C++ programming language and libquantum is implemented as a C programming language library. Q Language provides classes for basic quantum operations like QHadamard, QFourier, QNot, QSwap, which are derived from the base class Qop. New operators can be defined using C++ class mechanism. Both Q Language and libquantum share some limitation with QCL, since it is possible to operate on single qubits or quantum registers (*i.e.* arrays of qubits) only. Thus, they are similar to packages for computer algebra systems used to simulate quantum computation [85, 86].

Concerning problems with physical implementations of quantum computers, it became clear that one needs to take quantum errors into account when modelling quantum computational process. Also quantum communication has become very promising application of quantum information theory over the last few years. Both facts are reflected in the design of new quantum programming languages.

LanQ developed by Mlnařík was defined in [77, 87]. It provides syntax based on C programming language. LanQ provides several mechanisms such as the creation

of a new process by forking and interprocess communication, which support the implementation of multi-party protocols. Moreover, operational semantics of LanQ has been defined. Thus, it can be used for the formal reasoning about quantum algorithms.

It is also worth to mention new quantum programming languages based on functional paradigm. Research in functional quantum programming languages started by introducing quantum lambda calculus [88]. It was introduced in a form of simulation library for Scheme programming language. QPL [89] was the first functional quantum programming language. This language is statically typed and allows to detect errors at compile-time rather than run-time.

A more mature version of QPL is cQPL — communication capable QPL [75]. cQPL was created to facilitate the development of new quantum communication protocols. Its interpreter uses QCL as a backend language so cQPL programmes are translated into C++ code using QCL simulation library.

Table III contains the comparison of several quantum programming languages. It includes the most important features of existing languages. In particular we list the underlying mathematical model (*i.e.* pure or mixed states) and the support for quantum communication.

All languages listed in Table III are universal and thus they can be used to compute any function computable on a quantum Turing machine. Consequently, all these language provide the model of quantum computation which is equivalent to the model of a quantum Turing machine.

In this section we compare the selected quantum programming languages and provide some examples of quantum algorithms and protocols implemented in these languages. We also describe their main advantages and limitations. We introduce the basic syntax of three of the languages listed in Table III – QCL, LanQ and cQPL. This is motivated by the fact that these languages have a working interpreter and can be used to perform simulations of quantum algorithms. We introduce basic elements of QCL required to understand basic programmes. We also compare the main features of the presented languages.

The main problem with current quantum programming languages is that they tend to operate on very low-level structures only. In QCL quantum memory can be accessed using only `qreg` data type, which represents the array of qubits. In the syntax of cQPL data type `qint` has been introduced, but it is only synonymous for the array of 16 qubits. Similar situation exists in LanQ [87], where quantum data types are introduced using `q`n`it` keyword, where n represents a dimension of elementary unit (*e.g.* for qubits $n = 2$, for qutrits $n = 3$). However, only unitary evolution and measurement can be performed on variables defined using one of these types.

A. Requirements for quantum programming language

Taking into account QRAM model described in Section II C 2 we can formulate basic requirements which have to be fulfilled by any quantum programming language [74].

- **Completeness:** Language must allow to express any quantum circuit and thus enable the programmer to code every valid quantum programme written as a quantum circuit.
- **Extensibility:** Language must include, as its subset, the language implementing some high level classical computing paradigm. This is important since some parts of quantum algorithms (for example Shor's algorithm) require nontrivial classical computation.
- **Separability:** Quantum and classical parts of the language should be separated. This allows to execute any classical computation on purely classical machine without using any quantum resources.
- **Expressivity:** Language has to provide high level elements for facilitating the quantum algorithms coding.
- **Independence:** The language must be independent from any particular physical implementation of a quantum machine. It should be possible to compile a given programme for different architectures without introducing any changes in its source code.

As we will see, the languages presented in this Section fulfil most of the above requirements. The main problem is the *expressivity* requirement.

B. Imperative quantum programming

First we focus on quantum programming languages which are based on the imperative paradigm. They include quantum pseudocode, discussed in Section II C 2, Quantum Computation Language (QCL) created by Ömer [61, 81, 82] and LanQ developed by Mlnařík [76, 77, 87]

Below we provide an introduction to QCL. It is one of the most popular quantum programming languages. Next, we introduce the basic elements of LanQ. This language provides the support for quantum protocols. This fact reflects the recent progress in quantum communication theory.

1. Quantum Computation Language

QCL (Quantum Computation Language) [61, 81, 82] is the most advanced implemented quantum programming

	QCL	Q Language	QPL	cQPL	LanQ
reference	[81]	[84]	[89]	[75]	[87]
implemented	✓	✓	✓	✓	✓
formal semantics	–	–	✓	✓	✓
communication	–	–	–	✓	✓
universal	✓	✓	✓	✓	✓
mixed states	–	–	✓	✓	✓

Table III. The comparison of quantum programming languages with information about implementation and basic features. Based on information from [75] and [87].

language. Its syntax resembles the syntax of C programming language [90] and classical data types are similar to data types in C or Pascal.

The basic built-in quantum data type in QCL is **qureg** (quantum register). It can be interpreted as the array of qubits (quantum bits).

```
qureg x1[2]; // 2-qubit quantum register x1
qureg x2[2]; // 2-qubit quantum register x2
H(x1);       // Hadamard operation on x1
H(x2[1]); // and on the second qubit of x2
```

Listing 2. Basic operations on quantum registers and subregisters in QCL.

QCL standard library provides standard quantum operators used in quantum algorithms, such as:

- Hadamard H and Not operations on many qubits,
- controlled not (CNot) with many target qubits and Swap gate,
- rotations: RotX, RotY and RotZ,
- phase (Phase) and controlled phase (CPhase).

Most of them are described in Table II in Section II C 2.

Since QCL interpreter uses `qlib` simulation library, it is possible to observe the internal state of the quantum machine during the execution of quantum programmes. The following sequence of commands defines two-qubit registers a and b and executes H and CNot gates on these registers.

```
qcl> qureg a[2];
qcl> qureg b[2];
qcl> H(a);
[4/32] 0.5 |0,0> + 0.5 |1,0> +
 0.5 |2,0> + 0.5 |3,0>
qcl> dump
: STATE: 4 / 32 qubits allocated,
28 / 32 qubits free
0.5 |0> + 0.5 |1> + 0.5 |2> + 0.5 |3>
qcl> CNot(a[1],b)
[4/32] 0.5 |0,0> + 0.5 |1,0> + 0.5 |2,0>
+ 0.5 |3,0>
qcl> dump
: STATE: 4 / 32 qubits allocated,
28 / 32 qubits free
0.5 |0> + 0.5 |1> + 0.5 |2> + 0.5 |3>
```

Using **dump** command it is possible to inspect the internal state of a quantum computer. This can be helpful for checking if our algorithm changes the state of quantum computer in the requested way.

One should note that **dump** operation is different from measurement, since it does not influence the state of quantum machine. This operation can be realised using simulator only.

a. Quantum memory management Quantum memory can be controlled using quantum types **qureg**, **quconst**, **quvoid** and **quscratch**. Type **qureg** is used as a base type for general quantum registers. Other types allow for the optimisation of generated quantum circuit. The summary of types defined in QCL is presented in Table IV.

b. Classical and quantum procedures and functions QCL supports user-defined operators and functions known from languages like C or Pascal. Classical subroutines are defined using **procedure** keyword. Also standard elements, known from C programming language, like looping (*e.g.* **for** i=1 **to** n { ... }) and conditional structures (*e.g.* **if** x==0 { ... }), can be used to control the execution of quantum and classical elements. In addition to this, it provides two types of quantum subroutines.

The first type is used for unitary operators. Using it one can define new operations, which in turn can be used to manipulate quantum data. For example operator diffuse defined in Listing 3 defines *inverse about the mean* operator used in Grover's algorithm [14]. This allows to define algorithms on the higher level of abstraction and extend the library of functions available for a programmer.

Using subroutines it is easy to describe quantum algorithms. Figure 10 presents QCL implementation of Deutsch's algorithm, along with the quantum circuit for this algorithm. This simple algorithm uses all main elements of QCL. It also illustrates all main ingredients of existing quantum algorithms.

The second type of quantum subroutine is called a *quantum function*. Quantum functions are also called *pseudo-classic operators*. It can be defined using **qufunct** keyword. The subroutine of type **qufunct** is used for all transformations of the form

$$|n\rangle = |f(n)\rangle, \tag{17}$$

Type	Description	Usage
qureg	general quantum register	basic type
quvoid	register which has to be empty when operator is called	target register
quconst	must be invariant for all operators used in quantum conditions	quantum conditions
quscratch	register which has to be empty before and after the operator is called	temporary registers

Table IV. Types of quantum registers used for memory management in QCL.

where $|n\rangle$ is a base state and f is a one-to-one Boolean function. The example of quantum function is presented in Listing 5.

c. Quantum conditions QCL introduces *quantum conditional statements*, *i.e.* conditional constructions where quantum state can be used as a condition.

QCL, as well as many classical programming languages, provides the conditional construction of the form

if *be* **then**
block

where *be* is a Boolean expression and *block* is a sequence of statements.

QCL provides the means for using quantum variables as conditions. Instead of a classical Boolean variable, the variable used in condition can be a quantum register.

```
qureg a[2];
qureg b[2];
// the sequence of statements
// ...
// perform CNot if a=|1...1>
if a {
  CNot(b[0], b[1]);
}
```

Listing 4. Example of a quantum conditional statement in QCL

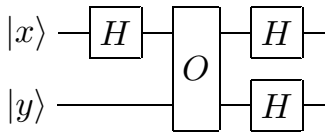
In this situation QCL interpreter builds and executes the sequence of $CNOT$ gates equivalent to the above condition. Here register a is called *enable register*.

In addition, quantum conditional structures can be used in quantum subroutines. Quantum operators and functions can be declared as conditional using **cond** keyword. For example

```
operator diffuse(qureg q) {
  H(q);              // Hadamard Transform
  Not(q);            // Invert q
  CPhase(pi,q);      // Rotate if q=1111..
  !Not(q);           // undo inversion
  !H(q);             // undo Hadamard Transform
}
```

Listing 3. The implementation of the inverse about the mean operation in QCL [61]. Constant pi represents number π. Exclamation mark ! is used to indicate that the interpreter should use the inverse of a given operator. Operation diffuse is used in the quantum search algorithm [14].

```
// conditional phase gate
extern cond operator Phase(real phi);
// conditional not gate
extern cond qufunct Not(qureg q);
```




```
operator U(qureg x,qureg y) {
  H(x);
  Oracle(x,y);
  H(x & y);
}

// classical control structure
procedure deutsch() {
  // allocate 2 qubits
  qureg x[1];
  qureg y[1];
  int m;
  // evaluation loop
  {
    // initialise machine state
    reset;
    // do unitary computation
    U(x,y);
    // measure 2nd register
    measure y,m;
    // value in 1st register valid?
  } until m==1;

  // measure 1st register which
  measure x,m;
  // contains g(0) xor g(1)
  print "g(0) xor g(1) =",m;
  // clean up
  reset;
}
```

Figure 10. Quantum circuit for Deutsch's algorithm and QCL implementation of this algorithm (see [61] for more examples). Evaluation loop is composed of preparation (performed by **reset** instruction), unitary evolution (U(x,y) operator) and measurement. Subroutine Oracle() implements function used in Deutsch's algorithm [42, 91].

declares a conditional Phase gate and a controlled *NOT* gate. Keyword **extern** indicates that the definition of a subroutine is specified in an external file. The enable register (*i.e.* quantum condition) is passed as an implicit parameter if the operator is used within the body of a quantum if-statement.

```
// increment register
cond qufunct inc(qureg x) {
  int i;
  for i = #x-1 to 0 step -1 {
  // apply controlled-not from MSB to LSB
    CNot(x[i],x[0::i]);
  }
}

// equivalent implementation
// with constant enable register
// conditional increment as selection operator
qufunct cinc(qureg x,quconst e) {
  int i;
  for i = #x-1 to 0 step -1 {
    CNot(x[i],x[0::i] & e);
  }
}
```

Listing 5. Operator for incrementing quantum state in QCL defined as a conditional quantum function. Subroutine inc is defined using **cond** keyword and does not require the second argument of type **quconst**. Subroutine cinc provides equivalent implementation with explicit-declared enable register.

In the case of inc procedure, presented in Listing 5, the enable register is passed as an implicit argument. This argument is set by a quantum if-statement and transparently passed on to all suboperators. As a result, all suboperators have to be conditional. This is illustrated by the following example [61]

```
// counting and control registers
qcl> qureg q[4];qureg e[1];
// prepare test state
qcl> H(q[3] & e);
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |0,1>
+ 0.5 |8,1>
// conditional increment
qcl> cinc(q,e);
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |1,1>
+ 0.5 |9,1>
// equivalent to cinc(q,e)
qcl> if e { inc(q); }
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |2,1>
+ 0.5 |10,1>
// conditional decrement
qcl> !cinc(q,e);
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |1,1>
+ 0.5 |9,1>
// equivalent to !cinc(q,e);
qcl> if e { !inc(q); }
[5/32] 0.5 |0,0> + 0.5 |8,0> + 0.5 |0,1>
+ 0.5 |8,1>
```

Finally we should note that a conditional subroutine can be called outside a quantum if-statement. In such situation enable register is empty and, as such, ignored. Subroutine call is in this case unconditional.

2. *LanQ*

Imperative language LanQ is the first quantum programming language with full operation semantics specified [87].

Its main feature is the support for creating multipartite quantum protocols. LanQ, as well as cQPL presented in the next section, are built with quantum communication in mind. Thus, in contrast to QCL, they provide the features for facilitating simulation of quantum communication.

Syntax of the LanQ programming language is very similar to the syntax of C programming language. In particular it supports:

- Classical data types: **int** and **void**.
- Conditional statements of the form

```
if ( cond ) {
  ...
} else {
  ...
}
```

- Looping with **while** keyword

```
while ( cond ) {
  ...
}
```

- User-defined functions, for example

```
int fun( int i) {
  int res;
  ...
  return res;
}
```

a. Process creation LanQ is built around the concepts of process and interprocess communication, known for example from UNIX operating system. It provides the support for controlling quantum communication between many parties. The implementation of teleportation protocol presented in Listing 6 provides an example of LanQ features, which can be used to describe quantum communication.

Function main() in Listing 6 is responssible for controlling quantum computation. The execution of protocol is divided into the following steps:

1. Creation of the classical channel for communicating the results of measurement:
   channel[**int**] c **withends** [c0,c1];.
2. Creation of Bell state used as a quantum channel for teleporting a quantum state
   (psiEPR **aliasfor** [psi1, psi2]); this is accomplished

by calling external function createEPR() creating an entangled state.

3. Instruction **fork** executes alice() function, which is used to implement sender; original process continues to run.

4. In the last step function bob() implementing a receiver is called.

```
void alice(channelEnd[int] c0,
           qbit auxTeleportState) {
  int i;
  qbit phi;
  // prepare state to be teleported
  phi = computeSomething();
  // Bell measurement
  i = measure (BellBasis, phi,
               auxTeleportState);
  send (c0, i);
}

void bob(channelEnd[int] c1,
         qbit stateToTeleportOn) {
  int i;
  i = recv(c1);
  // execute one of the Pauli gates
  // according to the protocol
  if (i == 1) {
    Sigma_z(stateToTeleportOn);
  } else if (i == 2) {
    Sigma_x(stateToTeleportOn);
  } else if (i == 3) {
    Sigma_x(stateToTeleportOn);
    Sigma_z(stateToTeleportOn);
  }
  dump_q(stateToTeleportOn);
}

void main() {
  channel[int] c withends [c0,c1];
  qbit psi1, psi2;
  psiEPR aliasfor [psi1, psi2];

  psiEPR = createEPR();

  c = new channel[int]();
  fork alice(c0, psi1);
  bob(c1, psi2);
}
```

Listing 6. Teleportation protocol implemented in LanQ [87]. Functions Sigma_x(), Sigma_y() and Sigma_z() are responsible for implementing Pauli matrices. Function createEPR() (not defined in the listing) creates maximally entangled state between parties — Alice and Bob. Quantum communication is possible by using the state, which is stored in a global variable psiEPR. Function computeSomething() (not defined in the listing) is responsible for preparing a state to be teleported by Alice.

b. Communication Communication between parties is supported by providing **send** and **recv** keywords. Communication is synchronous, *i.e.* **recv** delays programme execution until there is a value received from the channel and **send** delays a programme run until the sent value is received.

Processes can allocate *channels*. It should be stressed that the notion of channels used in quantum programming is different from the one used in quantum mechanics. In quantum programming a channel refers to a variable shared between processes. In quantum mechanics a channel refers to *any quantum operation*.

Another feature used in quantum communication is variable aliasing. In the teleportation protocol presented in Listing 6 the syntax for variable aliasing

```
qbit psi1, psi2;
psiEPR aliasfor [psi1, psi2];
```

is used to create quantum state shared among two parties.

c. Types Types in LanQ are used to control the separation between classical and quantum computation. In particular they are used to prohibit copying of quantum registers. The language distinguishes two groups of variables [87, Chapter 5]:

- Duplicable or non-linear types for representing classical values, *e.g.* **bit**, **int**, **boolean**. The value of a duplicable type can be exactly copied.
- Non-duplicable or linear types for controlling quantum memory and quantum resources, *e.g.* **qbit**, **qtrit** channels and channel ends (see example in Listing 6). Types from this group do not allow for cloning [92].

One should note that quantum types defined in LanQ are mainly used to check validity of the program before its run. However, such types do not help to define abstract operations. As a result, even simple arithmetic operations have to be implemented using elementary quantum gates, *e.g.* using quantum circuits introduced in [93].

C. Functional quantum programming – QPL and cQPL

During the last few years few quantum programming languages based on functional programming paradigm have been proposed [94]. As we have already point out, the lack of progress in creating new quantum algorithms is caused by the problems with operating on complex quantum states. Classical functional programming languages have many features which allow to clearly express algorithms [41]. In particular they allow for writing better modularised programmes than in the case of imperative programming languages [95]. This is important since this allows to debug programmes more easily and reuse software components, especially in large and complex software projects.

Quantum functional programming attempts to merge the concepts known from classical function programming with quantum mechanics. The program in functional programming language is written as a function, which is defined in terms of other functions. Classical functional programming languages contain no assignment statements, and this allows to eliminate side-effects.[96] It means that function call can have no effect other than to compute its result [95]. In particular it cannot change the value of a global variable.

The first attempts to define a functional quantum programming language were made by using quantum lambda calculus [88], which was based on lambda calculus. For the sake of completeness we can also point out some research on modelling quantum computation using Haskell programming language [97, 98]. However, here we focus on high-level quantum programming languages. Below we present recently proposed languages QPL and cQPL, which are based on functional paradigm. They aim to provide mechanisms known from programming languages like Haskell [99] to facilitate the modelling of quantum computation and quantum communication.

In [89] Quantum Programming Language (QPL) was described and in [75] its extension useful for modelling of quantum communication was proposed. This extended language was name cQPL – communication capable QPL. Since cQPL compiler is also QPL compiler, we will describe cQPL only.

The compiler for cQPL language described in [75] is built on the top of `libqc` simulation library used in QCL interpreter. As a result, cQPL provides some features known from QCL.

Classical elements of cQPL are very similar to classical elements of QCL and LanQ. In particular cQPL provides conditional structures and loops introduced with **while** keyword.

```
new int loop := 10;
while (loop > 5) do {
    print loop;
    loop := loop - 1;
};
if (loop = 3) then {
    print "loop is equal 3";
} else {
    print "loop is not equal 3";
};
```

Listing 7. Classical control structures in cQPL.

Procedures

Procedures can be defined to improve modularity of programmes.

```
proc test: a:int, q:qbit {
  ...
}
```

Procedure call has to know the number of parameters returned by the procedure. If, for example, procedure test is defined as above, it is possible to gather the calculated results

```
new int a1 = 0;
new int cv = 0;
new int qv = 0;
(a1) := call test(cv, qv);
```

or ignore them

```
call test(cv, qv);
```

In the first case the procedure returns the values of input variables calculated at the end of its execution.

Classical variables are passed by value *i.e.* their value is copied. This is impossible for quantum variable, since a quantum state cannot be cloned [92]. Thus, it is also impossible to assign the value of quantum variable calculated by procedure.

Note that no cloning theorem requires quantum variables to be *global*. This shows that in quantum case it is impossible to avoid some effects known from imperative programming and typically not present in functional programming languages.

Global quantum variables are used in Listing 9 to create a maximally entangled state in a teleportation protocol. Procedure createEPR(epr1, epr2) operates on two quantum variables (subsystems) and produces a Bell state.

Quantum elements

Quantum memory can be accessed in cQPL using variables of type **qbit** or **qint**. Basic operations on quantum registers are presented in Listing 8. In particular, the execution of quantum gates is performed by using $*=$ operator.

```
new qbit q1 := 0;
new qbit q2 := 1;
// execute CNOT gate on both qubits
q1, q2 *= CNot;
// execute phase gate on the first qubit
q1 *= Phase 0.5;
```

Listing 8. State initialisation and basic gates in cQPL. Data type `qbit` represents a single qubit.

It should be pointed out that **qint** data type provides only a shortcut for accessing the table of qubits.

Only a few elementary quantum gates are built into the language:

- Single qubit gates `H`, `Phase` and `NOT` implementing basic gates listed in Table II in Section II C 2.
- `CNOT` operator implementing controlled negation and `FT(n)` operator for n-qubit quantum Fourier transform.

This allows to simulate an arbitrary quantum computation. Besides, it is possible to define gates by specifying their matrix elements.

Measurement is performed using **measure**/**then** keywords and **print** command allows to display the value of a variable.

```
measure a then {
  print "a is |0>";
} else {
  print "a is |1>";
};
```

In similar manner like in QCL, it is also possible to inspect the value of a state vector using **dump** command.

Quantum communication

The main feature of cQPL is its ability to build and test quantum communication protocols easily. Communicating parties are described using *modules*. In analogy to LanQ, cQPL introduces channels, which can be used to send quantum data. Once again we stress that notion of channels used in cQPL and LanQ is different from that used in quantum theory. Quantum mechanics introduces channels to describe allowed physical transformations, while in quantum programming they are used to describe communication links.

Communicating parties are described by modules, introduced using **module** keyword. Modules can exchange quantum data (states). This process is accomplished using **send** and **receive** keywords.

To compare cQPL and LanQ one can use the implementation of the teleportation protocol. The implementation of teleportation protocol in cQPL is presented in Listing 9, while the implementation in LanQ is provided in Listing 6.

```
module Alice {
    proc createEPR: a:qbit, b:qbit {
       a *= H;
       b,a *= CNot;
       /* b: Control, a: Target */
    } in {
      new qbit teleport := 0;
      new qbit epr1 := 0;
      new qbit epr2 := 0;

      call createEPR(epr1, epr2);
      send epr2 to Bob;

      /* teleport: Control, epr1: Target
         (see: Figure 7)  */
      teleport, epr1 *= CNot;

      new bit m1 := 0;
      new bit m2 := 0;
      m1 := measure teleport;
      m2 := measure epr1;

      /* Transmit the classical
         measurement results to Bob */
      send m1, m2 to Bob;
};

module Bob {
   receive q:qbit from Alice;
   receive m1:bit, m2:bit from Bob;

   if (m1 = 1) then { q *= [[ 0,1,1,0 ]];
/* Apply sigma_x */ };

   if (m2 = 1) then { q *= [[ 1,0,0,-1 ]];
/* Apply sigma_z */};

   /* The state is now teleported */
   dump q;
};
```

Listing 9. Teleportation protocol implemented in cQPL (from [75]). Two parties – Alice and Bob – are described by modules. Modules in cQPL are introduced using **module** keyword.

IV. SUMMARY

The main goal of this paper is to acquaint the reader with quantum programming languages and computational models used in quantum information theory. We have described a quantum Turing machine, quantum circuits and QRAM models of quantum computation. We have also presented three quantum programming languages – namely QCL, LanQ and cQPL.

First we should note that the languages presented in this paper provide very similar set of basic quantum gates and allow to operate only on the arrays of qubits. Most of the gates provided by these languages correspond to the basic quantum gates presented in Section II C 2. Thus, one can conclude that the presented languages have the ability to express quantum algorithms similar to the abilities of a quantum circuit model.

The biggest advantage of quantum programming languages is their ability to use classical control structures for controlling the execution of quantum operators. This is hard to achieve in quantum circuits model and it requires the introduction of non-unitary operations to this model. In addition, LanQ and cQPL provide the syntax for clear description of communication protocols.

The syntax of presented languages resembles the syntax of popular classical programming languages from the C programming language family [90]. As such, it can be easily mastered by programmers familiar with classical languages. Moreover, the description of quantum algorithms in quantum programming languages is better suited for people unfamiliar with the notion used in quantum mechanics.

The main disadvantage of described languages is the lack of quantum data types. The types defined in described languages are used mainly for two purposes:

- To avoid compile-time errors caused by copying of quantum registers (cQPL and LanQ).
- Optimisation of memory management (QCL).

Both reasons are important from the simulations point of view, since they facilitate writing of correct and optimised quantum programmes. However, these features do not provide a mechanism for developing new quantum algorithms or protocols.

ACKNOWLEDGMENTS

The author would like to thank W. Mauerer for providing preliminary version of his cQPL compiler and acknowledge interesting discussions with P. Gawron, B. Ömer, I. Glendinning and H. Mlnařík. The author is also grateful to the anonymous referee for well formulated and interested remarks concerning the functional quantum programming languages.

This work was supported by the Polish National Science Centre under the grants number N N516 475440, N N516 481840 and by the Polish Ministry of Science and Higher Education under the grant number N N519 442339.

[1] D. Bouwmeester, A. Ekert, and A. Zeilinger (Eds.). *The Physics of Quantum Information: Quantum Cryptography, Quantum Teleportation, Quantum Computation*. Springer-Verlag, Berlin, Germany (2000).

[2] T. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. O'Brien. "Quantum computers". *Nature* 464, 45–53 (2010).

[3] G. E. Moore. "Cramming More Components Onto Integrated Circuits". *Electronics* 38 (8), 114–117 (1965).

[4] Intel Corporation. "Moores Law: Intel Microprocessor Transistor Count Chart" (2005). URL `http://intel.com/pressroom/kits/events/moores_law_40th/`.

[5] R. P. Feynman. "Simulating Physics with Computers". *Int. J. Theor. Phys.* 21 (6/7), 467–488 (1982).

[6] C. H. Bennett and G. Brassard. "Quantum cryptography: public key distribution and coin tossing". In *Proceedings of the IEEE International Conference on Computers, Systems, and Signal Processing, Bangalore, India*, pages 175–179 (1984).

[7] A. Ekert. "Quantum cryptography based on Bell's theorem". *Phys. Rev. Lett.* 67, 661–663 (1991).

[8] A. Einstein, B. Podolsky, and N. Rosen. "Can Quantum-Mechanical Description of Physical Reality Be Considered Complete?" *Phys. Rev.* 47 (10), 777–780 (1935).

[9] J. Bouda. *Encryption of Quantum Information and Quantum Cryptographic Protocols*. Ph.D. thesis, Faculty of Informatics, Masaryk University (2004).

[10] R. Ursin, F. Tiefenbacher, T. Schmitt-Manderbach, H. Weier, T. Scheidl, M. Lindenthal, B. Blauensteiner, T. Jennewein, J. Perdigues, P. Trojek, B. Ömer, M. Fürst, M. Meyenburg, J. Rarity, Z. Sodnik, C. Barbieri, H. Weinfurter, and A. Zeilinger. "Entanglement-based quantum communication over 144 km". *Nature Physics* 3, 481–486 (2007).

[11] M. Peev, C. Pacher, R. Alléaume, C. Barreiro, J. Bouda, W. Boxleitner, T. Debuisschert, E. Diamanti, M. Dianati, J. F. Dynes, S. Fasel, S. Fossier, M. Fürst, J. D. Gautier, O. Gay, N. Gisin, P. Grangier, A. Happe, Y. Hasani, M. Hentschel, H. Hübel, G. Humer, T. Länger, M. Legré, R. Lieger, J. Lodewyck, T. Lorünser, N. Lütkenhaus, A. Marhold, T. Matyus, O. Maurhart, L. Monat, S. Nauerth, J. B. Page, A. Poppe, E. Querasser, G. Ribordy, S. Robyr, L. Salvail, A. W. Sharpe, A. J. Shields, D. Stucki, M. Suda, C. Tamas, T. Themel, R. T. Thew, Y. Thoma, A. Treiber, P. Trinkler, R. Tualle-Brouri, F. Vannel, N. Walenta, H. Weier, H. Weinfurter, I. Wimberger, Z. L. Yuan, H. Zbinden, and A. Zeilinger. "The SECOQC quantum key distribution network in Vienna". *New Journal of Physics* 11 (7), 075001 (2009).

[12] P. W. Shor. "Algorithms for quantum computation: Discrete logarithms and factoring". In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE Computer Society Press (1994).

[13] P. Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computers". *SIAM J. Computing* 26, 1484–1509 (1997).

[14] L. K. Grover. "Quantum Mechanics Helps in Searching for a Needle in a Haystack". *Phys. Rev. Lett.* 79, 325–328 (1997).

[15] J. Eisert, M. Wilkens, and M. Lewenstein. "Quantum Games and Quantum Strategies". *Phys. Rev. Lett.* 83, 3077–3080 (1999).

[16] D. Meyer. *AMS Contemporary Mathematics: Quantum Computation and Quantum Information Science*, vol. 305, chap. Quantum games and quantum algorithms. American Mathematical Society, Providence, Rhode Island, U.S.A. (2000).

[17] J. Kempe. "Quantum random walks: An introductory overview". *Contemp. Phys.* 44 (4), 307–327 (2003).

[18] J. Kośík. "Two models of quantum random walk". *Cent. Eur. J. Phys.* 4, 556–573 (2003).

[19] P. W. Shor. "Progress in quantum algorithms". *Quantum Information Processing* 3 (1-5) (2004).

[20] S. J. Lomonaco and L. Kauffman. "Search for New Quantum Algorithms". Tech. Rep. F30602-01-2-0522, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF (2005). URL `http://www.cs.umbc.edu/~lomonaco/qis/DARPA-FINAL-RPT.pdf`.

[21] E. Bernstein and U. Vazirani. "Quantum Complexity Theory". *SIAM J. Computing* 26 (5), 1411–1473 (1997).

[22] L. Fortnow. "One complexity theorist's view of quantum computing". *Theor. Comput. Sci.* 292 (3), 597–610

(2003).

[23] E. Klarreich. "Playing by quantum rules". *Nature* 414, 244–245 (2001).

[24] C. F. Lee and N. F. Johnson. "Game-theoretic discussion of quantum state estimation and cloning". *Phys. Lett. A* 319 (5-6), 429–433 (2003).

[25] A. Ambainis. "Quantum walk algorithm for element distinctness". *SIAM Journal on Computing* 37, 210–239 (2007).

[26] A. M. Childs and J. M. Eisenberg. "Quantum algorithms for subset finding". *Quantum. Inf. Comput.* 5, 593 (2005).

[27] A. M. Childs and J. Goldstone. "Spatial search by quantum walk". *Phys. Rev. A* 70 (2), 022314.1 (2004).

[28] A. M. Childs. "Universal Computation by Quantum Walk". *Phys. Rev. Lett.* 102 (18), 180501 (2009).

[29] A. P. Hines and P. C. E. Stamp. "Quantum walks, quantum gates, and quantum computers". *Phys. Rev. A* 75 (6), 062321 (2007).

[30] A. Ambainis. "Quantum walks and their algorithmic applications". *Int. J. Quant. Inf.* 1, 507–518 (2003).

[31] M. Santha. "Quantum walk based search algorithms". In *5th Theory and Applications of Models of Computation (TAMC08), Xian, April 2008*, vol. 4978 of *LNCS*, pages 31–46 (2008).

[32] S. E. Venegas-Andraca. "Introduction to special issue: Physics and computer science – quantum computation and other approaches". *Math. Struct. Comp. Sci.* 20 (6), 995–997 (2010).

[33] M. Mosca and J. Smith. "Algorithms for Quantum Computers". In *Handbook of Natural Computing*, Springer Reference. Springer Verlag, Berlin, Germany (2011).

[34] A. Childs and W. van Dam. "Quantum algorithms for algebraic problems". *Rev. Mod. Phys.* 82 (1), 1–52 (2010).

[35] C. H. Papadimitriou. *Computational complexity.* Addison-Wesley Publishing Company (1994).

[36] S. A. Cook and R. A. Reckhow. "Time-bounded Random Access Machines". In *Proceeedings of the forth Annual ACM Symposium on Theory of Computing*, pages 73–80 (1973).

[37] J. C. Shepherdson and H. E. Strugis. "Computability of Recursive Functions". *J. ACM* 10 (2), 217–255 (1963).

[38] H. Vollmer. *Introduction to Circuit Complexity.* Springer-Verlag, Berlin, Heidelberg, Germany (1999).

[39] A. Church. "An unsolvable problem of elementary number theory". *American Journal of Mathematics* 58, 345–363 (1936).

[40] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, U.S.A., 2nd ed. (1996). URL `http://mitpress.mit.edu/sicp/`.

[41] J. C. Mitchell. *Concepts in programming languages.* Cambridge University Press, Cambridge, U.K. (2003).

[42] D. Deutsch. "Quantum theory, the Church-Turing principle and the universal quantum computer". *Proc. R. Soc. Lond. A* 400, 97 (1985).

[43] C. Bohm. "On a family of Turing machines and the related programming language". *ICC Bull.* 3, 187–194 (1964).

[44] S. Aaronson and G. Kuperberg. "Complexity ZOO" (2010). URL `http://qwiki.stanford.edu/index.php/Complexity_Zoo`.

[45] A. Yao. "Quantum Circuit Complexity". In *Proceedings of the* 34^{th} *IEEE Symposium on Foundations of Computer Science*, pages 352–360. IEEE Computer Society Press (1993).

[46] H. Nishimura and M. Ozawa. "Computational complexity of uniform quantum circuit families and quantum Turing machines". *Theor. Comput. Sci.* 276, 147–181 (2002).

[47] U. Vazirani. "A Survey of Quantum Complexity Theory". *Proc. Sympos. Appl. Math.* 58 (2002).

[48] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman (1979).

[49] U. Zwick. "Scribe notes of the course Boolean Circuit Complexity" (1995). URL `http://www.cs.tau.ac.il/~zwick/scribe-boolean.html`.

[50] M. Hirvensalo. *Quantum computing.* Springer-Verlag, Berlin, Germany (2001).

[51] D. Deutsch. "Quantum Computational Networks". *Proc. R. Soc. Lond. A* 425, 73 (1989).

[52] T. Toffoli. "Bicontinuous extension of reversible combinatorial functions". *Math. Syst. Theory* 14, 13–23 (1981).

[53] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge, U.K. (2000).

[54] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. "Elementary gates for quantum computation". *Phys. Rev. A* 52, 3457 (1995).

[55] D. Deutsch, A. Barenco, and A. Ekert. "Universality in Quantum Computation". *Proc. R. Soc. Lond.* 449 (1937), 669–677 (1995).

[56] V. V. Shende, I. L. Markov, and S. S. Bullock. "Minimal universal two-qubit controlled-NOT-based circuits". *Phys. Rev. A* 69, 062321 (2004).

[57] M. Möttönen, J. J. Vartiainen, V. Bergholm, and M. M. Salomaa. "Quantum Circuits for General Multiqubit Gates". *Phys. Rev. Lett.* 93 (13), 130502 (2004).

[58] J. J. Vartiainen, M. Mottonen, and M. M. Salomaa. "Efficient decomposition of quantum gates". *Phys. Rev. Lett.* 92, 177902 (2004).

[59] S. Aaronson and D. Gottesman. "Improved simulation of stabilizer circuits". *Phys. Rev. A* 70 (5), 052328 (2004).

[60] E. Knill. "Conventions for quantum pseudocode". Tech. Rep. LAUR-96-2724, Los Alamos National Laboratory (1996).

[61] B. Ömer. *Structured Quantum Programming.* Ph.D. thesis, Vienna University of Technology (2003).

[62] R. Nagarajan, N. Papanikolaou, and D. Williams. "Simulating and Compiling Code for the Sequential Quantum Random Access Machine". *Electronic Notes in Theoretical Computer Science* 170, 101–124 (2007).

[63] R. Cleve and D. P. DiVincenzo. "Schumacher's quantum data compression as a quantum computation". *Phys. Rev. A* 54 (4), 2636–2650 (1996).

[64] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2^{nd} ed. (2001).

[65] K. M. Svore, A. W. Cross, A. V. Aho, I. L. Chuang, and I. L. Markov. "Toward a Software Architecture for Quantum Computing Design Tools". In P. Selinger (Ed.), *Proceedings of the 2nd International Workshop on Quantum Programming Languages* (2004).

[66] K. M. Svore, A. W. Cross, I. L. Chuang, A. V. Aho, and I. L. Markov. "A Layered Software Architecture for Quantum Computing Design Tools". *Computer* 39 (1), 74–83 (2006).

[67] S. Bettelli. *Toward an architecture for quantum programming*. Ph.D. thesis, Università di Trento (2002).

[68] L. K. Grover. "Quantum computers can search rapidly by using almost any transformation". *Phys. Rev. Lett.* 80, 4329–4332 (1998).

[69] M. Mosca. *Quantum Computer Algorithms*. Ph.D. thesis, Wolfson College, University of Oxford (1999).

[70] C. Bennett and S. Wiesner. "Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states". *Phys. Rev. Lett.* 69, 2881–2884 (1992).

[71] G. Brassard, A. Broadbent, and A. Tapp. "Quantum Pseudo-Telepathy". *Found. Phys.* 35, 1877–1907 (2005).

[72] D. Bacon and W. van Dam. "Recent Progress in Quantum Algorithms". *Commun. ACM* 53 (2), 84–93 (2010).

[73] E. H. Knill and M. A. Nielsen. *Encyclopedia of Mathematics, Supplement III*, chap. Theory of quantum computation. Kluwer (2002).

[74] S. Bettelli, L. Serafini, and T. Calarco. "Toward an architecture for quantum programming". *Eur. Phys. J. D* 25 (2), 181–200 (2003).

[75] W. Mauerer. *Semantics and Simulation of Communication in Quantum Programming*. Master's thesis, University Erlangen-Nuremberg (2005).

[76] H. Mlnařík. *Operational Semantics and Type Soundness of Quantum Programming Language LanQ*. Ph.D. thesis, Masaryk University (2007).

[77] H. Mlnařík. "Semantics Of Quantum Programming Language LanQ". *Int. J. Quant. Inf.* 6 (1, Supp.), 733–738 (2008).

[78] S. Gay. "Quantum Programming Languages: Survey and Bibliography". *Math. Struct. Comput. Sci.* 16 (4) (2006).

[79] D. Unruh. "Quantum programming languages". *Informatik – Forschung und Entwicklung* 21 (1–2), 55–63 (2006).

[80] R. Rüdiger. "Quantum Programming Languages: An Introductory Overview". *Comput. J* 50 (2), 134–150 (2007).

[81] B. Ömer. *A Procedural Formalism for Quantum Computing*. Master's thesis, Vienna University of Technology (1998).

[82] B. Ömer. *Quantum Programming in QCL*. Master's thesis, Vienna University of Technology (2000).

[83] H. Weimer. "The C library for quantum computing and quantum simulation version 1.1.0" (2010). URL `http://www.libquantum.de/`.

[84] S. Bettelli, L. Serafini, and T. Calarco. "Toward an architecture for quantum programming". *Eur. Phys. J. D* 25 (2), 181–200 (2003).

[85] J. A. Miszczak and P. Gawron. "Numerical simulations of mixed states quantum computation". *Int. J. Quant. Inf.* 3 (1), 195–199 (2005).

[86] P. Gawron, J. Klamka, J. Miszczak, and R. Winiarczyk. "Extending scientic computing system with structural quantum programming capabilities". *Bull. Pol. Acad. Sci.-Tech. Sci.* 58 (1), 77–88 (2010).

[87] H. Mlnařík. "LanQ – Operational Semantics of Quantum Programming Language LanQ". Tech. Rep. FIMU-RS-2006-10, Masaryk University (2006).

[88] A. van Tonder. "A Lambda Calculus for Quantum Computation". *SIAM J. Comput.* 33 (5), 1109–1135 (2004).

[89] P. Selinger. "Towards a Quantum Programming Language". *Math. Struct. Comput. Sci.* 14 (4), 527–586 (2004).

[90] B. W. Kernighan and D. M. Ritchie. *C Programming Language*. Prentice Hall, Upper Saddle River, N.J., U.S.A., 2nd ed. (1988).

[91] D. Deutsch and R. Jozsa. "Rapid solution of problems by quantum computation". *Proc Roy Soc Lond A* 439, 553–558 (1992).

[92] W. K. Wootters and W. H. Zurek. "A single quantum cannot be cloned". *Nature* 299, 802–803 (1982).

[93] V. Vedral, A. Barenco, and A. Ekert. "Quantum networks for elementary arithmetic operations". *Phys. Rev. A* 54, 147–153 (1996).

[94] P. Selinger. "A Brief Survey of Quantum Programming Languages". In *Proceedings of the 7th International Symposium on Functional and Logic Programming*, vol. 2998 of *LNCS*, pages 1–6 (2004).

[95] J. Hughes. "Why Functional Programming Matters". *Comput. J.* 32 (2), 98–107 (1989).

[96] This is true in *pure functional* programming languages like Haskell.

[97] A. Sabry. "Modeling quantum computing in Haskell". In *ACM SIGPLAN Haskell Workshop* (2003).

[98] J. Karczmarczuk. "Structure and interpretation of quantum mechanics: a functional framework." In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 50–61. ACM Press (2003).

[99] G. Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, U.K. (2007).